CRT-2001-19

Centre de recherche sur les transports (C.R.T.)
Centre for Research on Transportation

BALANCING ASSEMBLY LINES WITH MULTI-ATTRIBUTED TASKS

by

Sophie D. Lapierre
Angel Ruiz

May 2001

102-02939/

PUBLICATION
CRT-2001-19

BALANCING ASSEMBLY LINES WITH MULTI-ATTRIBUTED TASKS*

by

Sophie Lapierre[1]
Angel Ruiz[2]

* We wish to thank Ms Nathalie Gélinas, Novaxa Consulting, for helping us to design this software, and the Natural Sciences and Engineering Research Council of Canada (NSERC) for their financial support.

1 Centre de recherche sur les transports, Université de Montréal, C.P. 6128, succursale Centre-ville, Montréal, Canada H3C 3J7, and Département de mathématiques et de génie industriel, École Polytechnique, C.P. 6079, succursale Centre-ville, Montréal, Canada H3C 3A7 (sophiel@crt.umontreal.ca)

2 Centre de recherche sur les transports, Université de Montréal, C.P. 6128, succursale Centre-ville, Montréal, Canada H3C 3J7, and UT Compiègne, B.P. 20529, 60205 Compiègne Cedex, France (angel@crt.umontreal.ca)

Dépôt légal – Bibliothèque nationale du Québec
Bibliothèque nationale du Canada, 2001

Résumé

Nous présentons une heuristique pour l'équilibrage d'une chaîne d'assemblage où chaque tâche requiert un positionnement spécifique de l'opérateur par rapport au côté et à la hauteur du convoyeur. Nous avons testé la performance de l'algorithme sur un problème de taille industriel comptant 248 tâches, 400 contraintes d'antériorité et 4 attributs par tâche. Nous montrons que les heuristiques basées sur des régles de priorité peuvent être facilement adaptées afin de tenir compte des particularités que l'on retrouve dans le milieu industriel. Nous tenons à préciser que les contraintes de positionnement rendent le problème d'équilibrage plus difficile à résoudre. Le bon dosage de logique et du hasard dans les règles de priorité nous a permis d'obtenir des résultats très satisfaisants. En moins de 40 secondes l'algorithme implanté sur MS-ACCESS 97 génère de bonnes solutions au problème test.

Mots-clés : chaîne d'assemblage, heuristiques.

Abstract

This paper presents the performance of a priority-based heuristic to solve the balancing of an assembly line with two sides and two different heights. We have tested such heuristic on a complex problem having 248 tasks, 400 precedence constraints, and 4 task attributes. We show that priority-based heuristics can easily account for the practical aspects of industrial applications. However, the multi-attributed tasks and workstations increase the difficulty to balance the assembly line. The heuristics have to make good use of logic and randomness in its implementation to generate good solutions. The heuristic takes 40 seconds to give good solutions implemented on MS-ACCESS 97.

Keywords : assembly line, heuristics.

1 Introduction

Balancing assembly lines is a hard problem arising frequently in manufacturing. The basic problem consists, given a set of tasks to be processed, in finding an assignment of tasks to workstations such that the total number of stations is minimized. The problem is constrained by a set of precedence relations between the tasks and by the *cycle time*, the given maximum available time per workstation. Also, the industrial setting might bring other specifications, such as two-sided assembly lines, increasing the complexity of balancing an assembly line.

A Canadian manufacturer of appliances presented us the following industrial case :

1. The products move along a linear conveyor and workers can be placed on each sides;
2. Due to products size, some tasks can be performed only on a specific side of the conveyor;
3. Some tasks must be completed on the lower part of the product while others must be done on the top side (in order to keep a minimum degree of comfort, the conveyor has two different heights to facilitate the assembly of each type of tasks).

When assigning tasks to workstation, one must be careful to match tasks and workstation attributes. For example, Figure 1 presents a problem of ten tasks with side and height attributes. An optimal of 5 workstations is also shown in this figure for a cycle time of 10, the tasks grouped with dotted lines representing a workstation. Note that each task in a given workstation must have the same set of attributes. The attributes make it harder to find a good balancing solution. Using the same example, it would have taken only 3 workstations to assemble the product without attributes (A,C,F; B,D,E; G,H,I).

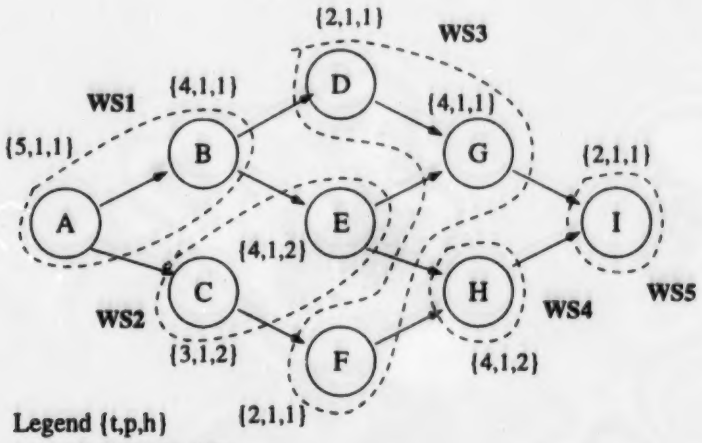

Figure 1: Assembly line with multi-attributed tasks

This paper is structured as follows. In section 2, we discuss the different algorithms proposed by the literature to balance an assembly line. In section 3 we present our algorithm and its implementation. Finally, we present the performance of our algorithm to solve a real industrial problem in section 4.

2 Literature Review

The balancing line problem has been extensively treated by the literature. Ghosh and Gagnon (1984) found more than 150 papers published on this problem in their comprehensive survey of the problem. Salveson (1955) was the first to present a mathematical formulation for the balancing line problem. Early work focused on the development of good heuristics for industrial problems. Arcus (1964) extensive work on the practical problems of balancing led him to the development of COMSOAL, a computerized software using what we refer as "priority-based" heuristics. This heuristic simply dispatches available tasks, those which precedent constrained tasks have been already assigned, according to a predefined priority rule such as "largest task first". Typically, different dispatching rules are tried on the same balancing problem since the most successful rule depends on the application problem (Boctor,1995). But randomness is also used by COMSOAL to explore a large number of solutions.

The research performed in the 70's and 80's almost exclusively focused on the development of exact methods to solve large instances of the basic assembly line problem. Fable, developed by Johnson (1988) is, up to this day, the best-performing exact algorithm to this NP-hard problem (Garey and Johnson, 1979). However, most balancing algorithms developed for specific industrial settings use heuristics (Ghosh and Gagnon ,1989) because they provide good solutions, and they do not require complex computer coding and software maintenance. Therefore, despite great advance on the theoretical level, most recent papers on the balancing problem in industrial settings are extensions of the heuristics developed by Arcus (1964).

Little work has been performed on the development of metaheuristics. Widner (1991) presents a generic metaheuristic with Tabu Search to solve the balancing problem. Chiang (1998) presents a refined version of a metaheuristic with Tabu Search on the generic balancing problem. It seems more work needs to be done on the development of metaheuristics for the balancing problem.

Work on the balancing problem with multi-attributed tasks is not very common. Bartholdi (1993) presents a good heuristic to solve the assembly problem with one-attributed tasks. He solves a problem coming from the automotive industry where assembly lines have workstations on both side of a linear conveyor. His algorithm solves the case of parallel workstations, as shown in Figure 2.a. No other author has worked on any extension of this problem: balancing the multi-attributed task problem. The multi-attributed task problem brings additional complexity to Bartholdi's problem. Work on that direction could lead to the development of much more flexible algorithms to solve industrial balancing problems. This paper focuses on a new industrial problem, an extension to Bartholdi's work: the two sided conveyor problem, the non-parallel workstations case as shown in Figure 2.b, with two different heights.

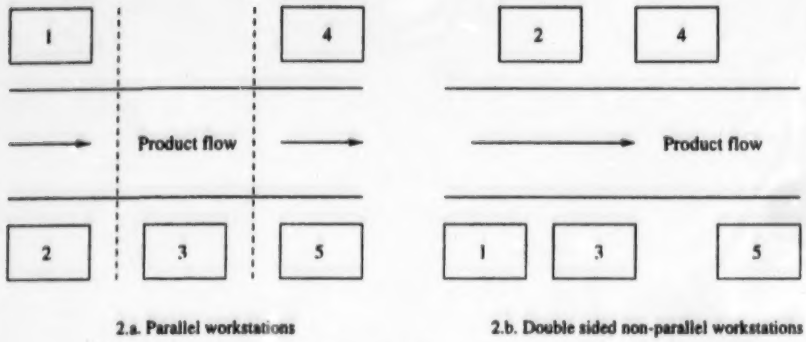

Figure 2: Two-sided assembly lines

3 Balancing with multi-attributed tasks

The balancing problem with multi-attributed tasks has several features that can be easily accounted by a "priority-based heuristics". Through the development and implementation of our heuristic, we found the problem harder to solve than expected. The consideration for attributes of tasks and workstations led us to the pseudocode presented in Figure 3. Let's define two lists named $TASK[i]$ and $WKST[j]$ containing respectively the following information on the tasks i and workstation j:

$TASK[i]$

- i : index for the tasks, $i = 1, \ldots, I$;
- *time* : time to execute task i;
- j : number of the workstation where task i has been allocated;
- *done* : $\begin{cases} 1 & \text{if task } i \text{ has been allocated to a workstation;} \\ 0 & \text{otherwise;} \end{cases}$
- *posi* : position attribute for task i (1= front task, 2= back task, 3 = either);
- *heig* : height attribute for task i (1= up workstation, 2= down workstation, 3= either).

$WKST[j]$

- j : index for the workstations, $j = 1, \ldots, J$;
- *time* : remaining available time at workstation j;
- *posi* : position attribute for workstation j (1= front workstation, 2= back workstation);
- *heig* : height attribute for task i (1= up workstation, 2= down workstation).

Let us also define another list named $A_TSK[k]$, a dynamic list which has the same structure as $TASK[i]$ but contains only the available tasks ready to be assigned to a workstation. The reason to have such a list is that the number of available tasks during the computing is very small compared to the total number of tasks. Since we need to sort the tasks each time a task is released or added to this list, this smaller list speeds up the execution.

The algorithm in Figure 3 starts by initializing (or resetting) the variables and the lists with their respective sorting criterias. The subroutine SEARCH_ FOR_AVAILABLE_

```
MAIN

1   Initialize
2   SEARCH_FOR_AVAILABLE_TASKS()
3   at ← COUNT_AVAILABLE_TASKS()
4   INIT_FIRST_WKS()
5   do while at > 0
6      sf, sb ← 0
7      ALLOCATE_FRONT(sf)
8      ALLOCATE_BACK(sb)
9      if sf = 0 and sb = 0 then
10        OPEN_NEW_STATION()
11     at ← COUNT_AVAILABLE_TASKS()
12  end do
```

Figure 3: Pseudocode of the multi-attributed tasks allocation algorithm

TASKS() then identifies the tasks without precedence constraints and moves them to the available tasks list A_TASK[i]. The number of available tasks in A_TASK[i] is then assigned to the variable at. Because we have a two-sided assembly line, the algorithm opens one front and one back worksation at the beginning of the execution (INIT_FIRST_WKS()). The height attribut for each station is selected based on the tasks available in the A_TASK[i] list. The algorithm then runs its main routine which is executed till all tasks i have been allocated to workstations (or till no available task remains). More precisely, the algorithm runs alternatively the allocation routines ALLOCATE_FRONT(sf) and ALLOCATE_BACK(sb), the variable sf and sb being used to manage properly the opening of a new workstation only when no available tasks can be assigned. When no task has been assigned either to the front or the back workstation, because the stations reached capacity or no tasks are available with the right attributes, the procedure OPEN_NEW_STATION() is called.

3.1 ALLOCATE_FRONT(sf) and ALLOCATE_BACK(sb)

The tasks are allocated to the workstations with the procedures ALLOCATE_FRONT(sf) and ALLOCATE_BACK(sb). Pseudocode for the allocation procedure ALLOCATE_FRONT(sf) is described in Figure 4. The procedure ALLOCATE_BACK(sb) is similar except that only tasks with the position attribute "2" or "3" are accepted, i.e., tasks requiring the back side of the conveyor or without a position requirement, and the number of the workstation jf is replaced by jb.

Before describing the procedure, we just want to remind that the available tasks are ranked by a pre-defined priority rule. Since we have tasks which are more flexible in their assignment, we give to all the tasks with position attribute "3" a lower priority level than the tasks with position attribute "1" or "2". Then, the procedure tries to allocate sequentially available tasks to workstations by a "first-fit" strategy. More precisely, after initializing the variables n and $stop$, the procedure examines the tasks sequentially on the list A_TASK until it finds a task that has the right position attribute. If such task is found, the procedure CHECK(A_TASK[n].i,jf) is called to verify if the task A_TASK[n].i can be added to the

```
ALLOCATE_FRONT(s1)
1   n ← 1
2   Stop ← 0
3   do while Stop = 0 and n ≤ at
4      if A_TASK[n].posi = 1 or 3 then
5         Stop =CHECK(A_TASK[n].i,jf)
6      if Stop = 1 then
7         TASK[A_TASK[n].i].done = 1
8         TASK[A_TASK[n].i].j = jf
9         A_TASK = A_TASK − {A_TASK[n]}
10        SEARCH_NEW_AVAILABLE_TASKS (A_TASK[n].i)
11        s1 ← 1
12     n ← n + 1
13  end do
```

Figure 4: Pseudocode of the front side allocation routine

current front workstation jf.

If a task is allocated to the current workstation, it is removed from the A_TASK list and its attribute TASK.*done* is set to "1". In the meantime, the workstation number jf is stored in the attribute j of TASK. The list A_TASK is then updated by calling the procedure SEARCH_NEW_AVAILABLE_TASKS(A_TASK[n].i) which looks if there are new tasks free of precedence constraints by the allocation of task A_TASK[n].i. The list A_TASK is automatically re-sorted according to the priority criteria. The variable "S1" is then set to 1 to tell the MAIN algorithm that a task has been successfully allocated. If all the tasks in the list A_TASK are assessed but none is allocated, the variable $s1$ remains at the value "0" to tell the MAIN algorithm so.

Pseudocode of the procedure CHECK(i,j) is shown in Figure 5. First, it ensures that only the tasks matching the position and height values of an open workstation become candidates to be allocated to i. Second, it checks if there is enough remaining time at the workstation to complete the task. Each time that a task is allocated to a workstation, the algorithm verifies if any new task becomes available and updates the ranking of the tasks. The procedure returns "1" if the task i can be allocated to the workstation j and "0" otherwise.

```
CHECK(i,j)
1   x ← 0
2      if TASK[i].posi = WKST[j].posi then
3         if TASK[i].heig = WKST[j].heig then
4            if WKST[j].time+TASK[i].time ≤ Cycle_Time then
5               x ← 1
6   return x
```

Figure 5: Pseudocode of the "fit" subroutine in multi-attributed tasks allocation algorithm

3.2 OPEN_NEW_STATION()

The procedure OPEN_NEW_STATION() is called when no tasks have been assigned to either the front or the back workstation. This procedure therefore decides on which side of the coveyer the station needs to be closed and another one opened, and which height attribute should be assigned to it. If all available tasks in A_TASK have the same position attribute, it is clear that no task were assigned because there was not enough remaining time in the station: the station with that position attribute is closed and another one is opened. If A_TASK has tasks with the two different attributes, we found no way to smartly decide which one to close: we pick randomly which one to close, with an equal probability to choose the back or front station.

Once the position for the new workstation has been decided, OPEN_NEW_STATION() has to define the workstation height attribute. It is done by looking at the available tasks that match the position of the new workstation: if all the tasks have the same height attribute, its value is chosen for the new workstation attribute. However, if there exist both up and down available tasks matching the new workstation position attribute, OPEN_NEW_STATION() selects a height value randomly with probability 1/2. After extensive test on our practical problem, this stochastic approach seems to offer a better performance than any other "smart" heuristic.

4 Experimental Results

We have tested our algorithm on a test problem inspired by a real problem coming from our industrial partner. The assembly problem has 248 tasks with 400 precedence constraints. Statistics on the tasks are summarized in Table 1. Note that only a few tasks have an "indifferent" attribute, a feature that makes our assembly problem even more different than the one presented by Bartholdi (1993).

	Attribute Height			
Attribute Position	Up	Down	Up or Down	**Total**
Front	50	95	3	148
Back	38	49	7	94
Front or Back	3	2	1	6
Total	91	146	11	**248**

Table 1: Task characteristics

We have two main concerns when we implement our algorithms: the performance of our algorithm to find good solutions and the time it takes to find it.

Since we needed to develop a software that would be user-friendly and could be easily maintained, we decided to implement our algorithm on Microsoft Visual Basic embedded in Access97. We have stored all data in an Access97 database, facilitating the maintenance of the software but also to take advantage of its sorting and querying capabilities to speed up the algorithm and also reduce the coding efforts. Finally, we have developed a window type user interface in Access97, with the addition of a graph generator software developed by Lucent Technologies to facilitate the validation of precedence constraints between the tasks

(see Lapierre and Ruiz (1999) for more details concerning the implementation). Generating the precedence constraints for the assembly of a product with 248 tasks is quite complex. The user interface was key to the success of our algorithm in the industrial setting.

The algorithm shown in Figure 3 is only the canvas for the implementation. In order to make it functional, we need to find the right number of iterations and the choice of dispatching rules. We have implemented four of the most popular dispatching rules: largest task first, shortest task first, task with the most successive tasks first, and random priority. The best dispatching rules depend on the application problem (Boctor 1995). For our industrial problem, we compare in Figure 6 the best results (out of 5 iterations) from each dispatching rule for different cycle times. The rule with "the task with the most successive tasks first" (called assembly sequence on the figure) was almost always the best solution and dispatching strategy that gives priority to the shortest tasks is consistently the worst. However, we kept all four rules in our implementation to add robustness to our algorithm in other settings.

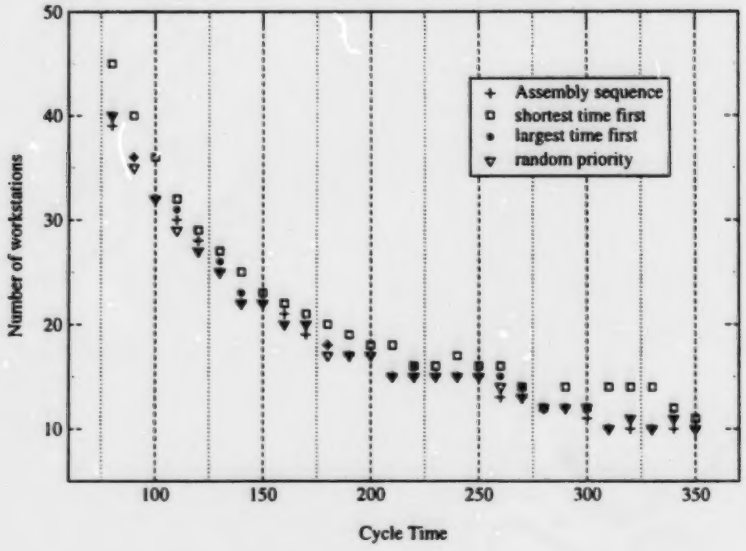

Figure 6: **Results for different cycle times**

The random selection of tasks does not give very good solutions compared to the other dispatching rules. However, for the selection of the workstation attributes, we found out that the random selection of the attributes gives better solutions than any "smart" selection rules we tried (most tasks with a given set of attributes and most tasks times with a given set of attributes). We elected to keep only the random selection for the workstation attributes.

The final implementation is the following: for each dispatching rule, 5 iterations are repeated in order to ensure the random effect on the *OPEN_NEW_STATION* procedure, for a total of 20 iterations. The execution time on a Pentium-II 266Mhz computer is less

than 40 seconds for a complete run. We have solved the problem provided by our industrial partner for several cycle times. We show, in Figure 7 our best solutions against the theoretical lower bound calculated by dividing the total manufacturing time by the cycle time. We can see that our algorithm gives solutions close to the optimal, between 85% and 92%. We can also see that we obtain better performance for certain cycle times. This is due to the integer nature of the problem but also to the structure of the constraints between the tasks. It is therefore important to optimize for several nearby cycle times (we did not implement this duty but trained accordingly the people using the software).

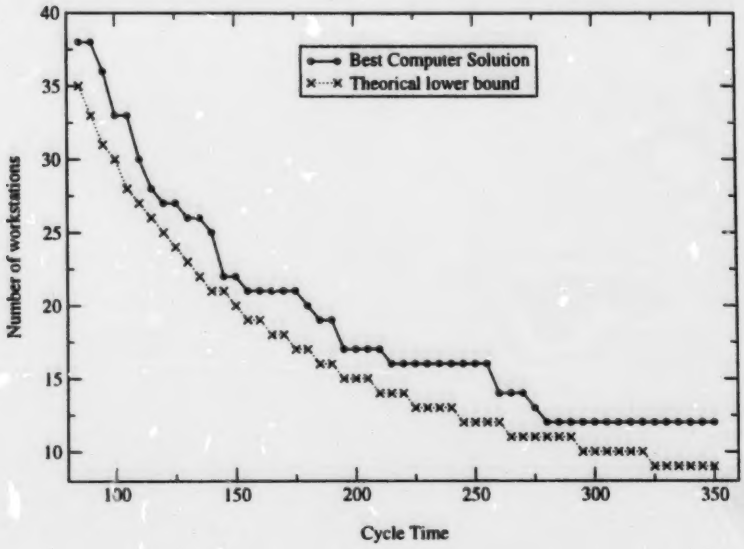

Figure 7: Results for different cycle times

Also, note that, in Figure 7 the performance of the algorithm decreases when cycle time increases. For cycle times of 280 seconds and more, no improvement is reached. The reason for this poorer performance is that large cycle times require another strategy to assemble the products: a very flexible workstation allowing repositioning of the product. We have introduced an option in our algorithm that, when activated, allows one change of the height attribute per workstation. This means that once per cycle time, we can ask a worker to climb a step (or go down a step), which is acceptable for long cycle time. At the algorithmic level, this means that we systematically try to change the height attribute of a workstation before trying to open a new station. Results obtained using this option are compared with the standard implementation and the theoretical solutions in Figure 8. We can see that better performance can be reached this way. However, we have not explored more elaborate search strategies for flexible context, such as exploring the change of attributes any time during the execution of the algorithm, not only when no tasks are available. More research

in that direction is needed, particularly since new assembly lines are designed to easily move the product on the conveyor.

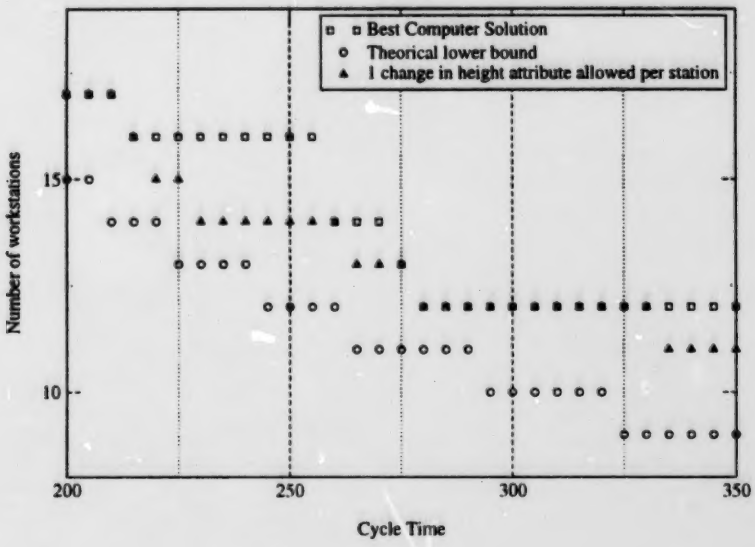

Figure 8: Quality of the solutions

5 Conclusion

We have developed a custom software package solving the balancing and management of complex assembly lines. We have shown how a simple heuristic approach offers a good performance in a real industrial case. Our heuristic fits all the customer requirements and it can be easily modified to handle new additional constraints. We have also detected the impact of the selection of attributes for a new station : we have concluded that a random choice performs better that no other deterministic rule. However, there is still room for further research especially regarding the balancing of flexible lines (or multi-attributed workstations) and finding the optimal cycle time for a given number of workstations. But, based on this case, we doubt that the priority based algorithms are suitable to this challenge. The metaheuristics should probably be explored for these more complex environments.

References

Arcus, A.L., (1966). "COMSOAL: a computer method of sequencing operations for assembly lines."*International Journal of Production Research*, **4**, 259–277.

Bartholdi, J.J., (1993). "Balancing two-sided assembly lines: a case study," *International Journal of Production Research*, **31**, 2447–2461.

Boctor, F.F., (1995). "A multiple-rule heuristic for assembly line balancing."*Journal of the Operational Research Society*, **46**, 62–69.

Chiang, W.C., (1998). "The application of a tabu search metaheuristic to the assembly line balancing problem."*Annals of Operations Research*, **77**, 209–227.

Garey, M.R. and Johnson, D.S., (1979). "Computers and intractability : a guide to the theory of NP-completeness," (San Francisco : W.H. Freeman).

Ghosh, S. and Gagnon, R.J., (1989). "A comprehensive literature review and analysis of the design, balancing and scheduling of assembly lines," *International Journal of Production Research*, **27**, 637–670.

Gutjahr, A. L., and Nemhauser, G. L., (1964). "An algorithm for the line balancing problem," *Management Science*, **11**, 308–315.

Hackman, S.T., Magazine, M.J. and Wee, T.S., (1989). "Fast, effective algorithms for simple assembly line balancing problems," *Operations Research* **37**, 916–924.

Johnson, R. V., (1988). "Optimally balancing large assembly lines with FABLE," *Management Science*, **34**, 240–253.

Lapierre, S. D. and Ruiz, A., (1999). "Balancement d'une ligne d'assemblage avec ACCESS97," *Proceedings of 3rd International Industrial Engineering Conference*Presses Internationales Polytechnique, Montreal, 357–364.

Widmer, M., (1991). *Modèles mathématiques pour une gestion efficace des ateliers flexibles*, Presses Polytechniques et Universitaires Romandes.